Discovery and Hot Replacement of Replicated Read-Only File Systems, with Application to Mobile Computing

Erez Zadok and Dan Duchamp
Computer Science Department
Columbia University

ABSTRACT

We describe a mechanism for replacing files, including open files, of a read-only file system while the file system remains mounted; the act of replacement is transparent to the user. Such a "hot replacement" mechanism can improve fault-tolerance, performance, or both. Our mechanism monitors, from the client side, the latency of operations directed at each file system. When latency degrades, the client automatically seeks a replacement file system that is equivalent to but hopefully faster than the current file system. The files in the replacement file system then take the place of those in the current file system. This work has particular relevance to mobile computers, which in some cases might move over a wide area. Wide area movement can be expected to lead to highly variable response time, and give rise to three sorts of problems: increased latency, increased failures, and decreased scalability. If a mobile client moves through regions having partial replicas of common file systems, then the mobile client can depend on our mechanism to provide increased fault tolerance and more uniform performance.

1 Introduction

The strongest trend in the computer industry today is the miniaturization of workstations into portable "notebook" or "palmtop" computers. Wireless network links [3] and new internetworking technology [8] offer the possibility that computing sessions could run without interruption even as computers move, using information services drawn from an infrastructure of (mostly) stationary servers.

We contend that operation of mobile computers according to such a model will raise problems that require re-thinking certain issues in file system design.[1] One such issue is how to cope with a client that moves regularly yet unpredictably over a wide area.

Several problems arise when a client moves a substantial distance away from its current set of servers. One is worse latency, since files not cached at the client must be fetched over longer distances. Another problem is increased probability of loss of connectivity, since gateway failures often lead to partitions. The final problem is decreased overall system "scalability:" more clients moving more data over more gateways means greater stress on the shared network.

One obvious way to mitigate these problems is to ensure that a file service client uses "nearby" servers at all times. A simple motivating example is that if a computer moves from New York to Boston, then in many cases it is advantageous to switch to using the Boston copies of "common" files like those in */usr/ucb*. As the client moves, the file service must be able to provide service first from one server, then from another. This switching mechanism should require no action on the

[1]Examples of such re-thinking can be found in [25] and [26].

part of administrators (since presumably too many clients will move too often and too quickly for administrators to track conveniently) and should be invisible to users, so that users need not become system administrators.

We have designed and implemented just such a file system — it adaptively discovers and mounts a "better" copy of a read-only file system which is fully or partially replicated. We define a better replica to be one providing better latency. Running our file service gives a mobile client some recourse to the disadvantages mentioned above. Our mechanism monitors file service latencies and, when response becomes inadequate, performs a dynamic attribute-guided search for a suitable replacement file system.

Many useful "system" file systems — and almost all file systems that one would expect to be replicated over a wide area — are typically exported read-only. Examples include common executables, manual pages, fonts, include files, etc. Indeed, read-only areas of the file space are growing fast, as programs increase the amount of configuration information, images, and on-line help facilities.

Although our work is motivated by the perceived needs of mobile computers that might roam over a wide area and/or frequently cross between public and private networks, our work can be useful in any environment characterized by highly variable response time and/or high failure rates.

Note that for a client to continue use of a file system as it moves, there must be underlying network support that permits the movement of a computer from one network to another without interruption of its sessions. Several such schemes have been developed [8, 7, 28, 29].

The remainder of this paper is organized as follows. In order to make a self-contained presentation, Section 2 provides brief explanations of other systems that we use in constructing ours. Section 3 outlines our design and Section 4 evaluates the work. Lastly, we mention related work in Section 5 and summarize in Section 6.

2 Background

Our work is implemented in and on SunOS 4.1.2. We have changed the kernel's client-side NFS implementation, and outside the operating system we have made use of the Amd automounter and the RLP resource location protocol. Each is explained briefly below.

2.1 NFS

Particulars about the NFS protocol and implementation are widely known and published [20, 12, 9, 19]. For the purpose of our presentation, the only uncommon facts that need to be known are:

- Translation of a file path name to a vnode is done mostly within a single procedure, called `au_lookuppn()`, that is responsible for detecting and expanding symbolic links and for detecting and crossing mount points.
- The name of the procedure in which an NFS client makes RPCs to a server is `rfscall()`.

We have made substantial alterations to `au_lookuppn()`, and slight alterations to `rfscall()`, `nfs_mount()`, `nfs_unmount()` and `copen()`.[2] We added two new system calls: one for controlling and querying the added structures in the kernel, and the other for debugging. Finally, we added fields to three major kernel data structures: vnode and vfs structures and the open file table.

2.2 RLP

We use the RLP resource location protocol [1] when seeking a replacement file system. RLP is a general-purpose protocol that allows a site to send broadcast or unicast request messages asking either of two questions:

[2] `Copen()` is the common code for `open()` and `create()`.

1. Do you (recipient site) provide this service?
2. Do you (recipient site) know of any site that provides this service?

A service is named by the combination of its transport service (e.g., TCP), its well-known port number as listed in `/etc/services`, and an arbitrary string that has meaning to the service. Since we search for an NFS-mountable file system, our RLP request messages contain information such as the NFS transport protocol (UDP [16]), port number (2049) and service-specific information such as the name of the root of the file system.

2.3 Amd

Amd [15] is a widely-used automounter daemon. Its most common use is to demand-mount file systems and later unmount them after a period of disuse; however, Amd has many other capabilities.

Amd operates by mimicking an NFS server. An Amd process is identified to the kernel as the "NFS server" for a particular mount point. The only NFS calls for which Amd provides an implementation are those that perform name binding: `lookup`, `readdir`, and `readlink`. Since a file must have its name resolved before it can be used, Amd is assured of receiving control during the first use of any file below an Amd mount point. Amd checks whether the file system mapped to that mount point is currently mounted; if not, Amd mounts it, makes a symbolic link to the mount point, and returns to the kernel. If the file system is already mounted, Amd returns immediately.

An example, taken from our environment, of Amd's operation is the following. Suppose `/u` is designated as the directory in which all user file systems live; Amd services this directory. At startup time, Amd is instructed that the mount point is `/n`. If any of the three name binding operations mentioned above occurs for any file below `/u`, then Amd is invoked. Amd consults its maps, which indicate that `/u/foo` is available on server `bar`. This file system is then mounted locally at `/n/bar/u/foo` and `/u/foo` is made a symbolic link to `/n/bar/u/foo`. (Placing the server name in the name of the mount point is purely a configuration decision, and is not essential.)

Our work is not dependent on Amd; we use it for convenience. Amd typically controls the (un)mounting of all file systems on the client machines on which it runs, and there is no advantage to our work in circumventing it and performing our own (un)mounts.

2.3.1 How Our Work Goes Beyond Amd

Amd does not already possess the capabilities we need, nor is our work a simple extension to Amd. Our work adds at least three major capabilities:

1. Amd keeps a description of where to find to-be-mounted file systems in "mount-maps." These maps are written by administrators and are static in the sense that Amd has no ability for automated, adaptive, unplanned discovery and selection of a replacement file system.
2. Because it is only a user-level automount daemon, Amd has limited means to monitor the response of `rfscall()` or any other kernel routine.
   Many systems provide a tool, like `nfsstat`, that returns timing information gathered by the kernel. However, `nfsstat` is inadequate because it is not as accurate as our measurements, and provides weighted average response time rather than measured response time. Our method additionally is less sensitive to outliers measures both short-term and long-term performance.
3. Our mechanism provides for transparently switching *open* files from one file system to its replacement.

3 Design

The key issues we see in this work are:

1. Is a switching mechanism really needed? Why not use the same file systems no matter where you are?
2. When and how to switch from one replica to another.
3. How to ensure that the new file system is an acceptable replacement for the old one.
4. How to ensure consistency if updates are applied across different replicas.
5. Fault tolerance: how to protect a client from server unavailability.
6. Security: NFS is designed for a local "workgroup" environment in which the space of user IDs is centrally controlled.

These issues are addressed below.

3.1 Demonstrating the Need

We contend that adaptive client-server matchups are desirable because running file system operations over many network hops is bad for performance in three ways: increased latency, increased failures, and decreased scalability. It is hard to ascertain exact failure rates and load on shared resources without undertaking a full-scale network study; however, we were able to gather some key data to support our claim. We performed a simple study to measure how latency increases with distance.

First, we used the *traceroute* program[3] to gather <*hop-count, latency*> data points measured between a host at Columbia and several other hosts around the campus, city, region, and continent. Latencies were measured by a Columbia host, which is a Sun-4/75 equipped with a microsecond resolution clock. The cost of entering the kernel and reading the clock is negligible, and so the measurements are accurate to a small fraction of a millisecond.

Next, we mounted NFS file systems that are exported Internet-wide by certain hosts. We measured the time needed to copy 1MB from these hosts using a 1KB block size. A typical result is plotted in Figure 1. Latency jumps by almost two orders of magnitude at the tenth hop, which represents the first host outside Columbia.

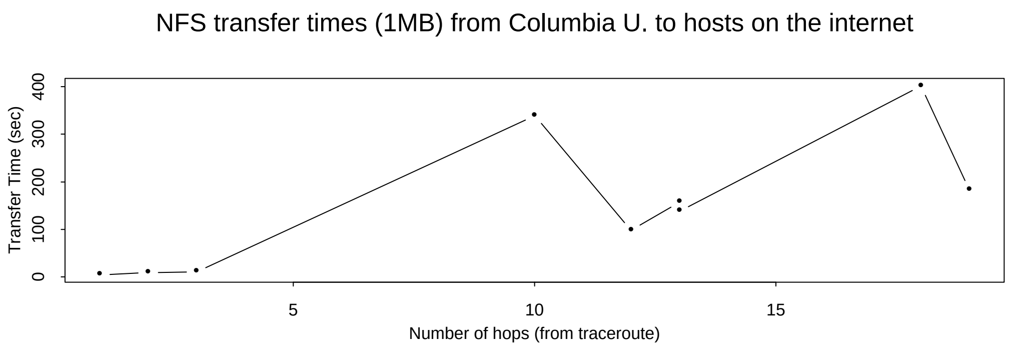

Figure 1: NFS Read Latency vs. Network Hop Count

[3]Written by Van Jacobson and widely available by anonymous ftp.

3.2 When to Switch

We have modified the kernel so that `rfscall()` measures the latency of every NFS `lookup` and maintains a per-filesystem data structure storing a number of recently measured latencies.

We chose to time the `lookup` operation rather than any other operation or mixture of operations for two reasons. The first is that `lookup` is the most frequently invoked NFS operation. We felt other calls would not generate enough data points to accurately characterize latency. The second reason is that `lookup` exhibits the least performance variability of the common NFS operations. Limiting variability of measured server latencies is important in our work, since we want to distinguish transient changes in server performance from long-term changes.

(At the outset of our work, we measured variances in the latency of the most common NFS operations and discovered huge swings, shown in Figure 2, even in an extended LAN environment that has been engineered to be uniform and not to have obvious bottlenecks. The measured standard deviations were 1027 msec for all NFS operations, 2547 msec for `read`, and 596 msec for `lookup`.)

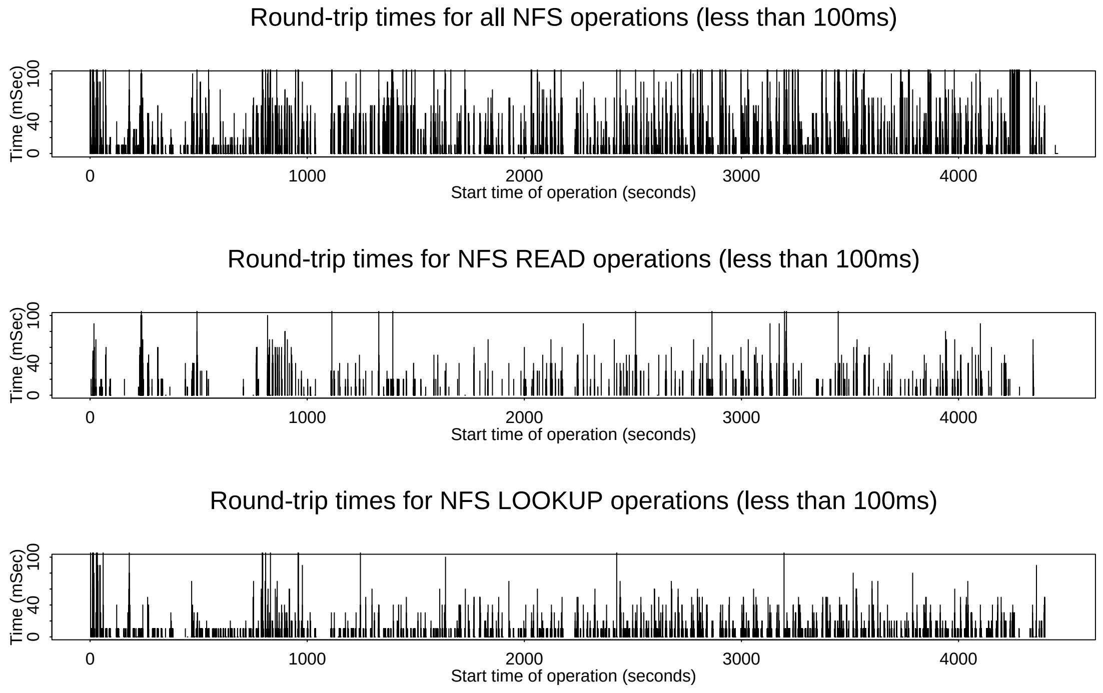

Figure 2: Variability and Latency of NFS operations

After addition of each newly measured `lookup` operation, the median latency is computed over the last 30 and 300 calls. We compute medians because medians are relatively insensitive to outliers. We take a data point no more than once per second, so during busy times these sampling intervals correspond to 30 seconds and 5 minutes, respectively. This policy provides insurance against anomalies like ping-pong switching between a pair of file systems: a file system can be replaced no more frequently than every 5 minutes.

The signal to switch is when, at any moment, the short-term median latency exceeds the long-term median latency by a factor of 2. Looking for a factor of two difference between short-term and long-term medians is our attempt to detect a change in performance which is substantial and "sudden," yet not transient. The length of the short-term and long-term medians as well as the ratio used to signal a switch are heuristics chosen after experimentation in our environment. All these parameters can be changed from user level through a debugging system call that we have added.

3.3 Locating a Replacement

When a switch is triggered, `rfscall()` starts a non-blocking RPC out to our user-level process that performs replacement, *nfsmgrd*.[4] The call names the guilty file server, the root of the file system being sought, the kernel architecture, and any mount options affecting the file system. *Nfsmgrd* uses these pieces of information to compose and broadcast an RLP request. The file system name keys the search, while the server name is a filter: the search must not return the same file server that is already in use.

The RLP message is received by the *nfsmgrd* at other sites on the same broadcast subnet. To formulate a proper response, an *nfsmgrd* must have a view of mountable file systems stored at its site and also mounted file systems that it is using — either type could be what is being searched for. Both pieces of information are trivially accessible through `/etc/fstab`, `/etc/exports`, and `/etc/mtab`.

The *nfsmgrd* at the site that originated the search uses the first response it gets; we suppose that the speed with which a server responds to the RLP request gives a hint about its future performance. (The Sun Automounter [2] makes the same assumption about replicated file servers.) If a read-only replacement file system is available, *nfsmgrd* instructs Amd to mount it and terminates the out-of-kernel RPC, telling the kernel the names of the replacement server and file system. The flow of control is depicted in Figure 3.

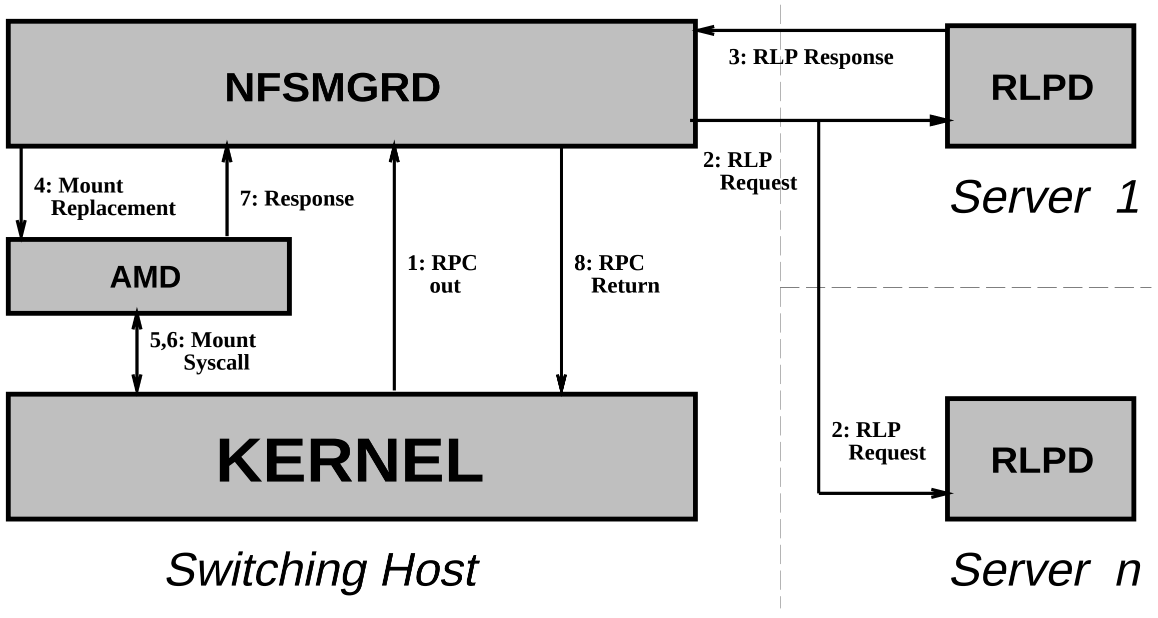

Figure 3: Flow of Control During a Switch

[4]Non-blocking operation is provided by a special kernel implementation of Sun RPC.

3.4 Using the Replacement

Once a replacement file system has been located and mounted, all future attempts to open files on the replaced file system will be routed to the replacement whenever they can be. Also, in all cases for which it is possible, *open files* on the replaced file system will be switched to their equivalents on the replacement. We describe these two cases in Sections 3.4.2 and 3.4.3, respectively.

3.4.1 Relevant Changes to Kernel Data Structures

In order to accommodate file system replacement, we have added some fields to three important kernel data structures: `struct vfs`, which describes mounted file systems; `struct vnode`, which describes open files; and `struct file`, which describes file descriptors.

The fields added to `struct vfs`, excluding debugging fields, are:

- The field `vfs_replaces` is valid in the vfs structure of the replacement file system; it points to the vfs structure of the file system being replaced.
- The field `vfs_replaced_by` is valid in the replaced file system's vfs struct; it points to the vfs structure of the replacement file system.
  When a replacement file system is mounted, our altered version of `nfs_mount()` sets the replaced and replacement file systems pointing to each other.
- The field `vfs_nfsmgr_flags` is valid for any NFS file system. One flag indicates whether the file system is managed by *nfsmgrd*; another indicates whether a file system switch is in progress.
- The field `vfs_median_info` contains almost all of the pertinent information about the performance of the file system, including the 300 most recent `nfs_lookup()` response times.
- The field `vfs_dft` is the *Duplicate File Table* (DFT). This per-filesystem table lists which files in the replacement file system have been compared to the corresponding file on the original file system mounted by Amd. Only equivalent files can be accessed on the replacement file system. The mechanism for making comparisons is described in Section 3.4.2.
  The size of the DFT is fixed (but changeable) so that new entries inserted will automatically purge old ones. This is a simple method to maintain "freshness" of entries.
  The DFT is a hash table whose entries contain a file pathname relative to the mount point, a pointer to the vfs structure of the replacement file system, and an extra pointer for threading the entries in insertion order. This data structure permits fast lookups keyed by pathname and quick purging of older entries.

The only field added to `struct vnode` is `v_last_used`, which contains the last time that `rfscall()` made a remote call on behalf of this vnode. This information is used in "hot replacement," as described in Section 3.4.3.

The only field added to `struct file` is `f_path`, which contains the relative pathname from the mount point to the file for which the descriptor was opened. Different entries may have different pathnames for the same file if several hard links point to the file.

3.4.2 After Replacement: Handling New Opens

When Amd mounts a file system it makes a symlink from the desired location of the file system to the mount point. For example, `/u/foo` would be a symlink pointing to the real mount point of `/n/bar/u/foo`; by our local convention, this would indicate that server `bar` exports `/u/foo`. Users and application programs know only the name `/u/foo`.

The information that `bar` exports a proper version of `/u/foo` is placed in Amd's mount-maps by system administrators who presumably ensure that the file system `bar:/u/foo` is a good version of whatever `/u/foo` should be. Therefore, we regard the information in the client's Amd mount-maps as authoritative, and consider any file system that the client might mount and place at `/u/foo` as a correct and complete copy of the file system. We call this file system *the master copy*, and use it for comparison against the replacement file systems that our mechanism locates and mounts.

The new open algorithm is shown in Figure 4. After a replacement has been mounted, whenever name resolution must be performed for any file on the replaced file system, the file system's DFT is first searched for the relative pathname. If the DFT indicates that the replacement file system has an equivalent copy of the file, then that file is used.

```
open() {
    examine vfs_replaced_by field to see if there is a replacement file system;
    if (no replacement file system) {
        continue name resolution;
        return;
    }
    if (DFT entry doesn't exist) {
        create and begin DFT entry;
        call out to perform file comparison;
        finish DFT entry;
    }
    if (files equivalent) {
        get vfs of replacement from vfs_replaces field;
        continue name resolution on replacement file system;
    } else
        continue name resolution on master copy;
}
```

Figure 4: New Open Algorithm

If the DFT contains an entry for the pathname, then the file on the replacement file system has already been compared to its counterpart on the master copy. A field in the DFT tells if the comparison was successful or not. If not, then the rest of the pathname has to be resolved on the master copy. If the comparison was successful, then the file on the replacement file system is used; in that case, name resolution continues at the root of the replacement file system.

If the DFT contains no entry for the pathname, then it is unknown whether the file on the replacement file system is equivalent to the corresponding file on the master copy.

To test equivalence, `au_lookuppn()` calls out of the kernel to *nfsmgrd*, passing it the two host names, the name of the file system, and the relative pathname to be compared. A partial DFT entry is constructed, and a flag in it is turned on to indicate that there is a comparison in progress and that no other process should initiate the same comparison.[5]

Nfsmgrd then applies, at user level, whatever tests might be appropriate to determine whether the two files are equivalent. This flow of control is depicted in Figure 5. Presently, we are performing file checksum comparison: *nfsmgrd* calls a *checksumd* daemon on each of the file servers, requesting the checksum of the file being compared. *Checksumd*, which we have written for this work, computes MD4 [18] file checksums on demand and then stores them for later use; checksums can also be pre-computed and stored.

Nfsmgrd collects the two checksums, compares them, and responds to the kernel, telling `au_lookuppn()` which pathname to use, always indicating the file on the replacement file system if possible. `Au_lookuppn()` completes the construction of the DFT entry, unlocks it, and marks which

[5]This avoids the need to lock the call out to *nfsmgrd*.

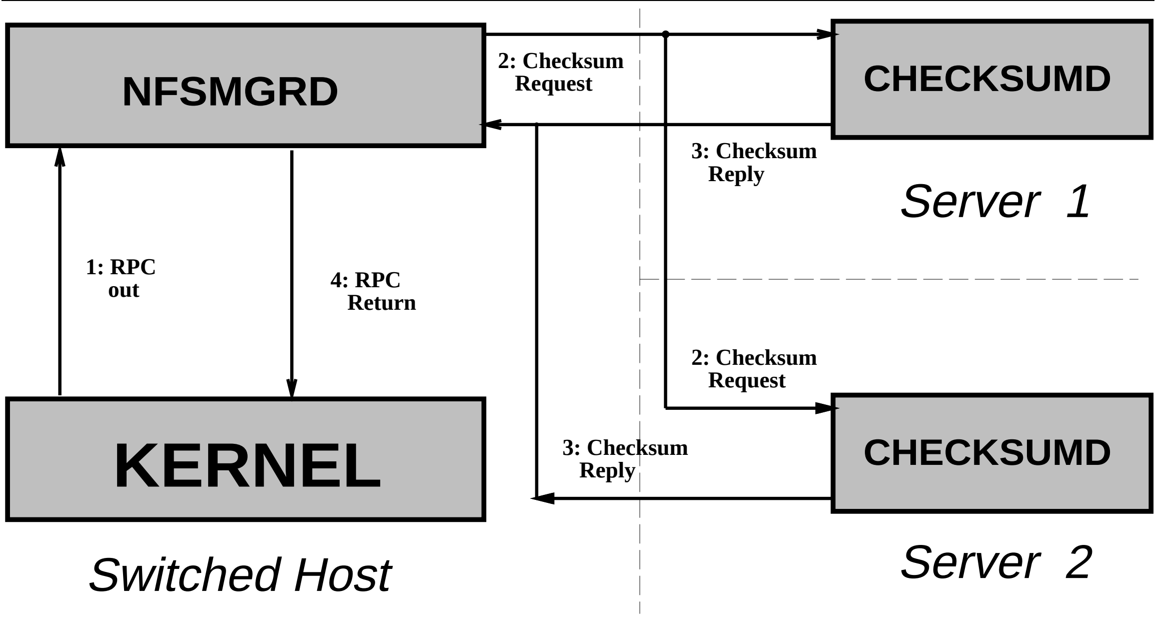

Figure 5: Flow of Control During File Comparison

vfs is the proper one to use whenever the same pathname is resolved again.

In this fashion, all new pathname resolutions are re-directed to the replacement file system whenever possible.

Note that the master copy could be unmounted (e.g., Amd by default unmounts a file system after a few minutes of inactivity), and this would not affect our mechanism. The next use of a file in that file system would cause some master copy to be automounted, before any of our code is encountered.

3.4.3 After Replacement: Handling Files Already Open

When a file system is replaced, it is possible that some files will be open on the replaced file system at the moment when the replacement is mounted. Were the processes with these open files to continue to use the replaced file system, several negative consequences might ensue. First, since the replacement is presumed to provide faster response, the processes using files open on the replaced file systems experience worse service. Second, since the total number of mounted file systems grows as replacements happen, the probability rises that some file system eventually becomes unavailable and causes processes to block. Further, the incremental effect of each successive file system replacement operation is reduced somewhat, since files that are open long-term do not benefit from replacement. Finally, kernel data structures grow larger as the number of mounted file systems climbs. Motivated by these reasons, we decided to switch *open files* from the replaced file system to the replacement file system whenever the file on the replacement file system is equivalent to that on the master copy.

Although this idea might at first seem preposterous, it is not, since we restrict ourselves to read-only file systems. We assume that files on read-only file systems[6] change very infrequently and/or are updated with care to guard against inconsistent reads.[7] Whether operating conditions uphold this assumption or not, the problem of a file being updated[8] while being read exists independently of our work, and our work does not increase the danger.

[6]That is, they are *exported* as read-only to some hosts, although they might be exported as read-write to others.
[7]An example of "careful update" is provided by the SUP utility [22].
[8]That is, updated by a host to which the file system is exported read-write.

We allow for a replacement file system to be itself replaced. This raises the possibility of creating a "chain" of replacement file systems. Switching vnodes from the old file system to its replacement limits this chain to length two (the master copy and the current replacement) in steady state.

The "hot replacement" code scans through the global open file table, keying on entries by vfs. Once an entry is found that uses the file system being replaced, a secondary scan locates all other entries using the same vnode. In a single entry into the kernel (i.e., "atomically"), all file descriptors pointing to that vnode are switched, thereby avoiding complex questions of locking and reference counting.

Hot replacement requires knowing pathnames. Thanks to our changes, the vfs structure records the pathname it is mounted on and identifies the replacement file system; also, the relative pathname of the file is stored in the file table entry. This information is extracted, combined with the host names, and passed out to *nfsmgrd* to perform comparison, as described above. If the comparison is successful, the pathname on the replacement file system is looked up, yielding a vnode on the replacement file system. This vnode simply replaces the previous vnode in all entries in the open file table. This results in a switch the next time a process uses an open file descriptor.

Hot replacement is enabled by the statelessness of NFS and by the vfs/vnode interfaces within the kernel. Since the replaced server keeps no state about the client, and since the open file table knows only a pointer to a vnode, switching this pointer in every file table entry suffices to do hot replacement.

An interesting issue is at which time to perform the hot replacement of vnodes. Since each file requires a comparison to determine equivalence, switching vnodes of all the open files of a given file system could be a lengthy process. The four options we considered are:

1. Switch as soon as a replacement file system is mounted (the early approach).
2. Switch only if/when an RPC for that vnode hangs (the late approach).
3. Switch if/when the vnode is next used (the "on-demand" approach).
4. Switch whenever a daemon instructs it to (the "flexible" approach).

The decision to switch earlier or later is affected by the tradeoff that early switching more quickly switches files to the faster file system and improves fault tolerance by reducing the number of file systems in use, but possibly wastes effort. Vnode switching is a waste in all cases when a vnode exists that will not be used again. Early switching also has the disadvantage of placing the entire delay of switching onto the single file reference that is unlucky enough to be the next one.

We chose the "flexible" approach of having a daemon make a system call into the kernel which then sweeps through the open file table and replaces some of the vnodes which can be replaced. We made this choice for three reasons. First, we lacked data indicating how long a vnode lingers after its final use. Second, we suspected that such data, if obtained, would not conclusively decide the question in favor of an early or late approach. Third, the daemon solution affords much more flexibility, including the possibility of more "intelligent" decisions such as making the switch during an idle period.

We emphasize that the system call into the kernel switches "some" of the vnodes, since it may be preferable to bound the delay imposed on the system by one of these calls. Two such bounding policies that we have investigated are, first, switching only N vnodes per call, and, second, switching only vnodes that have been accessed in the past M time units. Assuming that file access is bursty (a contention supported by statistics [14]), the latter policy reduces the amount of time wasted switching vnodes that will never be used again. We are currently using this policy of switching only recently used vnodes; this policy makes use of the `v_last_used` field that we added to the vnode structure.

3.5 Security

The NFS security model is the simple uid/gid borrowed from UNIX, and is appropriate only in a "workgroup" situation where there is a central administrative authority. Transporting a portable computer from one NFS user ID domain to another presents a security threat, since processes assigned user ID X in one domain can access exported files owned by user ID X in the second domain.

Accordingly, we have altered `rfscall()` so that every call to a replacement file system has its user ID and group ID both mapped to "nobody" (i.e., value -2). Therefore, only world-readable files on replacement file systems can be accessed.

3.6 Code Size

Counting blank lines, comments, and debugging support, we have written close to 11,000 lines of C. More than half is for user-level utilities: 1200 lines for the RLP library and daemon, 3200 for *nfsmgrd*, 700 lines for *checksumd*, and 1200 lines for a control utility (called *nfsmgr_ctl*). New kernel code totals 4000 lines, of which 800 are changes to SunOS, mostly in the NFS module. The remaining 3200 lines comprise the four modules we have added: 880 lines to deal with storing and computing medians; 780 lines are the "core NFS management" code, which performs file system switching, pathname storage and replacement, and out-of-kernel RPC; 540 lines to manage the DFT; and 1000 lines to support the `nfsmgr_ctl` system call.

The `nfsmgr_ctl` system call allows query and control over almost all data structures and parameters of the added facility. We chose a system call over a kmem program for security. This facility was used heavily during debugging; however, it is meant also for system administrators and other interested users who would like to change these "magic" variables to values more suitable for their circumstances.

4 Evaluation

This system is implemented and is receiving use on a limited number of machines.

The goal of this work is to improve overall file system performance — under certain circumstances, at least — and to improve it enough to justify the extra complexity. For this method to really work, it must have:

1. Low overhead latency measurement between switches.
2. A quick switch.
3. Low overhead access to the replacement after a switch.
4. No anomalies or instabilities, like ping-pong switching.
5. No process hangs due to server failures.
6. No security or administrative complications.

We have carried out several measurements aimed at evaluating how well our mechanism meets these goals.

The *overhead between switches* is that of performance monitoring. The added cost of timing every `rfscall()` we found too small to measure. The cost of computing medians could be significant, since we retain 300 values. But we implemented a fast incremental median algorithm that requires just a negligible fraction of the time in `nfs_lookup()`. The kernel data structures are not so negligible: retaining 300 latency measurements costs about 2KB per file system. The reason for the expansion is the extra pointers that must be maintained to make the incremental median algorithm work. The extra fields in the `struct vfs`, `struct vnode`, `struct file` are small, with the exception of the DFT, which is large. The current size of each (per-filesystem) DFT is 60 slots which occupy a total of 1KB-2KB on average.

Our measured *overall switch time* is approximately 3 sec. This is the time between the request for a new file system and when the new file system is mounted (messages 1-8 in Figure 3). Three seconds is comparable to the time needed in our facility to mount a file system whose location is already encoded in Amd's maps, suggesting that most of the time goes to the mount operation.

The *overhead after a switch* consists mostly of doing equivalence checks outside the kernel; the time to access the vfs of the replacement file system and DFT during `au_lookuppn()` is immeasurably small. Only a few milliseconds are devoted to calling *checksumd*: 5-7 msec if the checksum is already computed. This call to *checksumd* is done once and need not be done again so long as a record of equivalence remains in the DFT.

A major issue is how long to cache DFT entries that indicate equivalence. Being stateless, NFS does not provide any sort of server-to-client cache invalidation information. Not caching at all ensures that files on the replacement file system are always equal to those on the master copy; but of course this defeats the purpose of using the replacement. We suppose that most publicly-exported read-only file systems have their contents changed rarely, and thus one should cache to the maximum extent. Accordingly, we manage the DFT cache by LRU.

As mentioned above, *switching instabilities* are all but eliminated by preventing switches more frequently than every 5 minutes.

4.1 Experience

4.1.1 What is Read-Only

Most of the files in our facility reside on read-only file systems. However, sometimes one can be surprised. For example, GNU *Emacs* is written to require a world-writable lock directory. In this directory *Emacs* writes files indicating which users have which files in use. The intent is to detect and prevent simultaneous modification of a file by different processes. A side effect is that the "system" directory in which *Emacs* is housed (at our installation, `/usr/local`) must be exported read-write.

Deployment of our file service spurred us to change *Emacs*. We wanted `/usr/local` to be read-only so that we could mount replacements dynamically. Also, at our facility there are several copies of `/usr/local` per subnet, which defeats *Emacs*' intention of using `/usr/local` as a universally shared location. We re-wrote *Emacs* to write its lock files in the user's home directory since (1) for security, our system administrators wish to have as few read-write system areas as possible and, (2) in our environment by far the likeliest scenario of simultaneous modification is between two sessions of the same user, rather than between users.

4.1.2 Suitability of Software Base

Kernel. The vfs and vnode interfaces in the kernel greatly simplified our work. The hot replacement, in particular, proved far easier than we had feared, thanks to the vnode interface. The special out-of-kernel RPC library also was a major help. Nevertheless, work such as ours makes painfully obvious the benefits of implementing file service out of the kernel. The length and difficulty of the edit-compile-debug cycle, and the primitive debugging tools available for the kernel were truly debilitating.

RLP. RLP was designed in 1983, when the evils of over-broadcasting were not as deeply appreciated as they are today and when there were few multicast implementations. Accordingly, RLP is specified as a broadcast protocol. A more up-to-date protocol would use multicast. The benefits would include causing much less waste (i.e., bothering hosts that lack an RLP daemon) and contacting many more RLP daemons. Not surprisingly, we encountered considerable resistance from our bridges and routers when trying to propagate an RLP request. A multicast RLP request would travel considerably farther.

NFS. NFS is ill-suited for "cold replacement" (i.e., new opens on a replacement file system) caused by mobility, but is well suited for "hot replacement" because of its statelessness.

NFS' lack of cache consistency callbacks has long been bemoaned, and it affects this work since there is no way to invalidate DFT entries. Since we restrict ourselves to read-only files, the danger is assumed to be limited, but is still present. Most newer file service designs include cache consistency protocols. However, such protocols are not necessarily a panacea. Too much interaction between client and server can harm performance, especially if these interactions take place over a long distance and/or a low bandwidth connection. See [27] for a design that can ensure consistency with relatively little client-server interaction.

The primary drawback of using NFS for mobile computing is its limited security model. Not only can a client from one domain access files in another domain that are made accessible to the same user ID number, but even a well-meaning client cannot prevent itself from doing so, since there is no good and easy way to tell when a computer has moved into another uid/gid domain.

5 Related Work

It is a thesis of our work that in order for mobile computing to become the new standard model of computing, adaptive resource location and management will have to become an automatic function of distributed services software. The notion of constantly-networked, portable computers running modern operating systems is relatively new. Accordingly, we know of no work other than our own (already cited) on the topic of adaptive, dynamic mounting.

The Coda file system [21] supposes that mobile computing will take place in the form of "disconnected operation," and describes in [11] a method in which the user specifies how to "stash" (read/write) files before disconnection and then, upon reconnection, have the file service run an algorithm to detect version skew. Coda can be taken as a point of contrast to our system, since the idea of disconnection is antithetical to our philosophy. We believe trends in wireless communication point to the ability to be connected any time, anywhere. Users may *decide* not to connect (e.g., for cost reasons) but will not be *forced* not to connect (e.g., because the network is unreliable or not omnipresent). We call this mode of operation *elective connectivity*.

An obvious alternative to our NFS-based effort is to employ a file system designed for wide-area and/or multi-domain operation. Such file systems have the advantages of a cache consistency protocol and a security model that recognizes the existence of many administrative domains. Large scale file systems include AFS [6] and its spinoffs, Decorum [10] and IFS (Institutional File System) [5]. Experiments involving AFS as a "nation-wide" file service have been going on for years [23]. This effort has focused on stitching together distinct administrative domains so as to provide a single unified naming and protection space. However, some changes are needed to the present authentication model in order to support the possibility of a mobile client relocating in a new domain. In particular, if the relocated client will make use of local services, then there should be some means whereby one authentication agent (i.e., that in the new domain) would accept the word of another authentication agent (i.e., that in the client's home domain) regarding the identity of the client.

The IFS project has also begun to investigate alterations to AFS in support of mobile computers [4]. Specifically, they are investigating cache pre-loading techniques for disconnected operation and transport protocols that are savvy about the delays caused by "cell handoff" — the time during which a mobile computer moves from one network to another.

Plan 9's *bind* command has been designed to make it easy to mount new file systems. In particular, file systems can be mounted "before" or "after" file systems already mounted at the same point. The before/after concept replaces the notion of a search path. Plan 9 also supports the notion of a "union mount" [17]. The Plan 9 bind mechanism is a more elegant alternative to our double mounting plus comparison. However, a binding mechanism — even an unusually flexible one such as that of Plan 9 — addresses only part of the problem of switching between file systems. The harder part of the problem is determining *when* to switch and *what* to switch to.

6 Conclusion

We have described the operation, performance, and convenience of a transparent, adaptive mechanism for file system discovery and replacement. The adaptiveness of the method lies in the fact that a file service client no longer depends solely on a static description of where to find various file systems, but instead can invoke a resource location protocol to inspect the local area for file systems to replace the ones it already has mounted.

Such a mechanism is generally useful, but offers particularly important support for mobile computers which may experience drastic differences in response time as a result of their movement. Reasons for experiencing variable response include: (1) moving beyond the home administrative domain and so increasing the "network distance" between client and server and (2) moving between high-bandwidth private networks and low-bandwidth public networks (such movement might occur even within a small geographic area). While our work does not address how to access replicated read/write file systems or how to access one's home directory while on the move, our technique does bear on the problems of the mobile user. Specifically, by using our technique, a mobile user can be relieved of the choice of either suffering with poor performance or devoting substantial local storage to "system" files.[9] Instead, the user could rely on our mechanism to continuously locate copies of system files that provide superior latency, while allocating all or most of his/her limited local storage to caching or stashing read/write files such as those from the home directory.

Our work is partitioned into three modular pieces: heuristic methods for detecting performance degradation and triggering a search; a search technique coupled with a method for testing equivalence versus a master copy; and a method for force-switching open files from the use of vnodes on one file system to vnodes on another (i.e., "hot replacement"). There is little interrelationship among these techniques, and so our contributions can be viewed as consisting not just of the whole, but also of the pieces. Accordingly, we see the contributions of our work as:

1. The observation that file system switching might be needed and useful.
2. The idea of an automatically self-reconfiguring file service, and of basing the reconfiguration on measured performance.
3. Quantification of the heuristics for triggering a search for a replacement file system.
4. The realization that a "hot replacement" mechanism should not be difficult to implement in an NFS/vnodes setting, and the implementation of such a mechanism.

6.1 Future Work

There are several directions for future work in this area.

The major direction is to adapt these ideas to a file service that supports a more appropriate security model. One part of an "appropriate" security model is support for cross-domain authentication such that a party from one domain can relocate to another domain and become authenticated in that domain. Another part of an appropriate security model should include accounting protocols allowing third parties to advertise and monitor (i.e., "sell") the use of their exported file systems. Within the limited context of NFS, a small step in the right direction would be a mechanism that allows clients (servers) to recognize servers (clients) from a different domain. The most recent version of Kerberos contains improved support for cross-domain authentication, so another step in the right direction would be to integrate the latest Kerberos with NFS, perhaps as originally sketched in [24].

Another desirable idea is to convert from using a single method of exact file comparison (i.e., *checksumd*) to per-user, possibly inexact comparison. For example, object files produced by *gcc*

[9]One might suppose that a "most common subset" of system files could be designated and loaded, and this is true. However, specifying such a subset is ever harder as programs depend on more and more files for configuration and auxiliary information. This approach also increases the user's responsibility for system administration, which we regard as a poor way to design systems.

contain a timestamp in the first 16 bytes; two object files may be equal except for the embedded timestamps, which can be regarded as an insignificant difference. Another example is that data files may be equal except for gratuitous differences in floating-point format (e.g., 1.7 vs. 1.7000 vs. 1.70e01). Source files may be compared ignoring comments and/or white space. Intelligent comparison programs like *diff* or *spiff* [13] know how to discount certain simple differences.

Minor extensions to our work include: converting RLP from a broadcast protocol to a multicast protocol; and reimplementing in an environment (e.g., multi-server Mach 3.0) that supports out-of-kernel file service implementations.

7 Acknowledgements

We thank an anonymous member of the program committee for the suggestion to use file checksums. We thank the program committee, especially David Rosenthal and Matt Blaze, for valuable advice about presentation and emphasis.

This work was supported in part by the New York State Science and Technology Foundation's Center for Advanced Technology in Computers and Information Systems; by a National Science Foundation CISE Institutional Infrastructure grant, number CDA-90-24735; and by the Center for Telecommunications Research, an NSF Engineering Research Center supported by grant number ECD-88-11111.

8 References

[1] M. Accetta. Resource Location Protocol. RFC 887, IETF Network Working Group, December 1983.

[2] B. Callaghan and T. Lyon. The Automounter. In *Proc. 1989 Winter USENIX Conf.*, pages 43–51, January 1989.

[3] D. C. Cox. A Radio System Proposal for Widespread Low-power Tetherless Communication. *IEEE Trans. Communications*, 39(2):324–335, February 1991.

[4] P. Honeyman. Taking a LITTLE WORK Along. CITI Report 91-5, Univ. of Michigan, August 1991.

[5] J. Howe. Intermediate File Servers in a Distributed File System Environment. CITI Report 92-4, Univ. of Michigan, June 1992.

[6] J. H. Howard et al. Scale and Performance in a Distributed File System. *ACM Trans. Computer Systems*, 6(1):51–81, February 1988.

[7] J. Ioannidis et al. Protocols for Supporting Mobile IP Hosts Draft RFC, IETF Mobile Hosts Working Group, June 1992.

[8] J. Ioannidis, D. Duchamp and G. Q. Maguire Jr. IP-based Protocols for Mobile Internetworking. In *Proc. SIGCOMM '91*, pages 235–245. ACM, September 1991.

[9] C. Juszczak. Improving the Performance and Correctness of an NFS Server. In *Proc. 1989 Winter USENIX Conf.*, pages 53–63, January 1989.

[10] M. L. Kazar et al. Decorum File System Architectural Overview. In *Proc. 1990 Summer USENIX Conf.*, pages 151–163, June 1990.

[11] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. Computer Systems*, 10(1):3–25, February 1992.

[12] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun Unix. In *Proc. 1986 Summer USENIX Conf.*, pages 238–247, June 1986.

[13] D. Nachbar. Spiff – A Program for Making Controlled Approximate Comparisons of Files. In *Proc. 1986 Summer USENIX Conf.*, pages 238–247, June 1986.

[14] J. Ousterhout et al. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. Tenth ACM Symp. on Operating System Principles*, pages 15–24, December 1985.

[15] J. Pendry and N. Williams. *Amd - The 4.4 BSD Automounter.* Imperial College of Science, Technology, and Medicine, London, 5.3 alpha edition, March 1991.

[16] J. Postel. User Datagram Protocol. RFC 768, IETF Network Working Group, August 1980.

[17] D. Presotto et al. Plan 9, A Distributed System. In *Proc. Spring 1991 EurOpen Conf.*, pages 43–50, May 1991.

[18] R. Rivest. The MD4 Message-Digest Algorithm. RFC 1186, IETF Network Working Group, April 1992.

[19] D. S. H. Rosenthal. Evolving the Vnode Interface. In *Proc. 1990 Summer USENIX Conf.*, pages 107–117, June 1990.

[20] R. Sandberg et al. Design and Implementation of the Sun Network Filesystem. In *Proc. 1985 Summer USENIX Conf.*, pages 119–130, June 1985.

[21] M. Satyanarayanan et al. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Computers*, 39(4):447–459, April 1990.

[22] S. Shafer and M. R. Thompson. The SUP Software Upgrade Protocol. Unpublished notes available by ftp from mach.cs.cmu.edu:/mach3/doc/unpublished/sup/sup.doc

[23] A. Z. Spector and M. L. Kazar. Uniting File Systems. *UNIX Review*, 7(3):61–71, March 1989.

[24] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proc. 1988 Winter USENIX Conf.*, pages 191–202, February, 1988.

[25] C. Tait and D. Duchamp. Detection and Exploitation of File Working Sets. In *Proc. Eleventh Intl. Conf. on Distributed Computing Systems*, pages 2–9. IEEE, May 1991.

[26] C. Tait and D. Duchamp. Service Interface and Replica Consistency Algorithm for Mobile File System Clients. In *Proc. First Intl. Conf. on Parallel and Distributed Information Systems*, pages 190–197. IEEE, December 1991.

[27] C. Tait and D. Duchamp. An Efficient Variable Consistency Replicated File Service. In *File Systems Workshop*, pages 111–126. USENIX, May 1992.

[28] F. Teraoka, Y. Yokote, and M. Tokoro. A network Architecture Providing Host Migration Transparency. In *Proc. SIGCOMM '91*, pages 209–220. ACM, September 1991.

[29] H. Wada et al. Mobile Computing Environment Based on Internet Packet Forwarding. In *Proc. 1993 Winter USENIX Conf.*, pages 503–517, January 1993.

9 Author Information

Erez Zadok is an MS candidate and full-time Staff Associate in the Computer Science Department at Columbia University. This paper is a condensation of his Master's thesis. His primary interests include operating systems, file systems, and ways to ease system administration tasks. In May 1991, he received his B.S. in Computer Science from Columbia's School of Engineering and Applied Science. Erez came to the United States six years ago and has lived in New York "sin" City ever since. In his rare free time Erez is an amateur photographer, science fiction devotee, and rock-n-roll fan.

Mailing address: 500 West 120th street, Columbia University, New York, NY 10027. Email address: `ezk@cs.columbia.edu`.

Dan Duchamp is an Assistant Professor of Computer Science at Columbia University. His current research interest is the various issues in mobile computing. For his initial efforts in this area, he was recently named an Office of Naval Research Young Investigator.

Mailing address: 500 West 120th street, Columbia University, New York, NY 10027. Email address: `djd@cs.columbia.edu`.